System Design Document for Wack

Viktor Franzen, Tobias Lindroth, Spondon Siddiqui, Alexander Solberg

October 18, 2018

1 Introduction

This document is supposed to give the reader an deeper understanding about the design of our project, ”Wack”.

1.1 Design goals

The main design goal of this project is to have an easily extended application that is loosely coupled. It should be easily tested and the application should also have a clear MVC-structure.

1.2 Definitions, acronyms, and abbreviations

GUI: Graphical User Interface; How the application looks.

MVC: Model-View-Controller; A design pattern that separates the logic from the view which makes it easy to reuse the model or change view.

UML: Unified Modeling Language; A general-purpose, modeling language used in the development of object oriented projects. It provides a standard way for the design of a system to be visualized.

Channel: A chat group; A group where users of the application can join and send messages. Other users in the same group can see those messages.

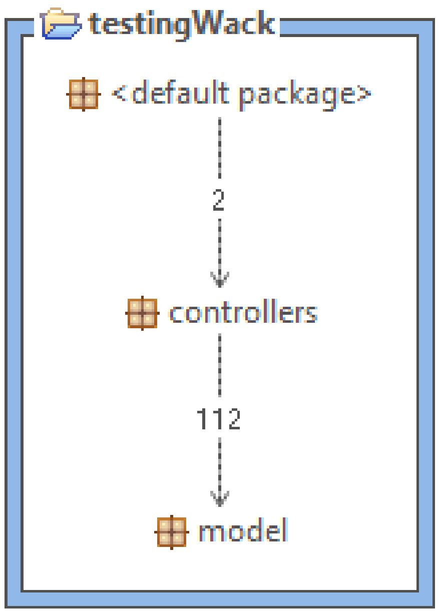
JSON: JavaScript Object Notation; A language independent data format.

2 System Architecture

This chat application will simulate an actual chat application with multiple computers. Hence, everything will be on the same computer, but using multiple clients.

When starting the program, there will be a login/sign up screen, from which the user can create clients. A client will open in a separate window and then the login/sign up screen can be used to create more clients.

In the top level of this application, we have the model, controllers and services packages and the main class (lies in ”default package”). As seen in the figure below, the controller and services package both uses the model, but the model is not aware of who is using it.



Although not showing in the figure, the controller package also uses the resource package to retrieve fxml files. The controllers and the fxml files are loosely coupled in the way that the controllers are set dynamically and the fxml files just expect an object with the correct methods.

In this way a MVC pattern is used. The model is not aware of what entity is using its data and logic. The controllers act as a connection between the model and its respective view. In this implementation the controller interacts with a facade towards the model and uses the information it receives to update the views. The separate views are built in fxml and hence, they don’t handle any logic, they just show what they are told to show. This means that both the controllers and the separate views can be replaced without it affecting the model packages.

2.1 Subsystem Decomposition

Our chat application is divided into 3 major packages which can be seen in "figure x". Model, Controllers, and Services See the appendix to see the general relationships between the packages and a fully expanded image of the packages.

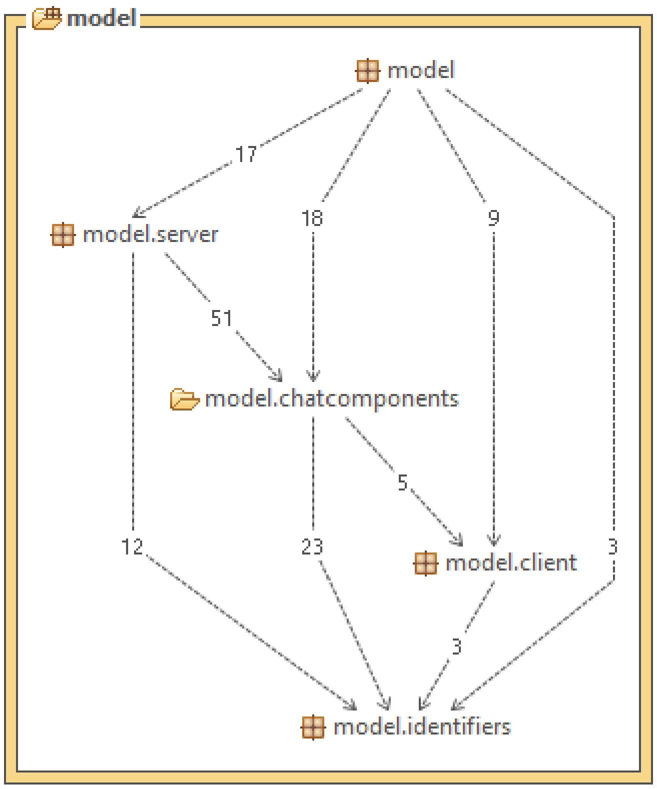
Model contains all the data and logic which is used to run the application.

Controller handles inputs and requests from view and communicates with the model.

Services are used by the controllers

2.2 Model

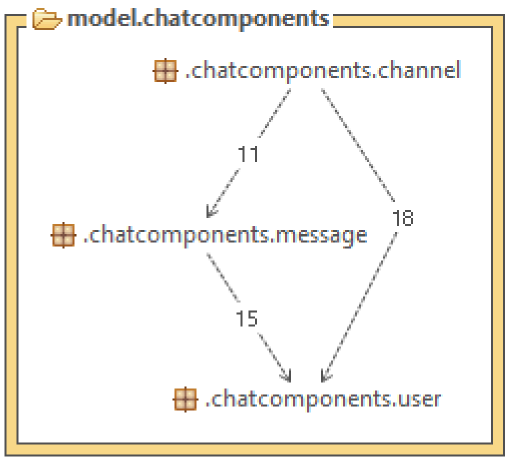
The model component are responsible for all the data and logic of the program. The top level of the model contains the ChatFacade and four packages that are called server, client, chatcomponents and identifiers.



The model package at the top is really just the facade of the model, called ChatFacade. This is responsible for providing a facade for the model package, to minimize the dependencies on model implementations.

The client package is responsible for being able to receive updates and provide an interface for anyone that want to add themselves as listener to also receive the updates the client receives. The client will then forward it to all it's listeners.

The "chatcomponents" package is responsible for providing the objects that creates the structure of the chat system. This package is divided into three subpackages, channel, messages and user.



The channel package is responsible of providing a component where several users can send messages to each other.

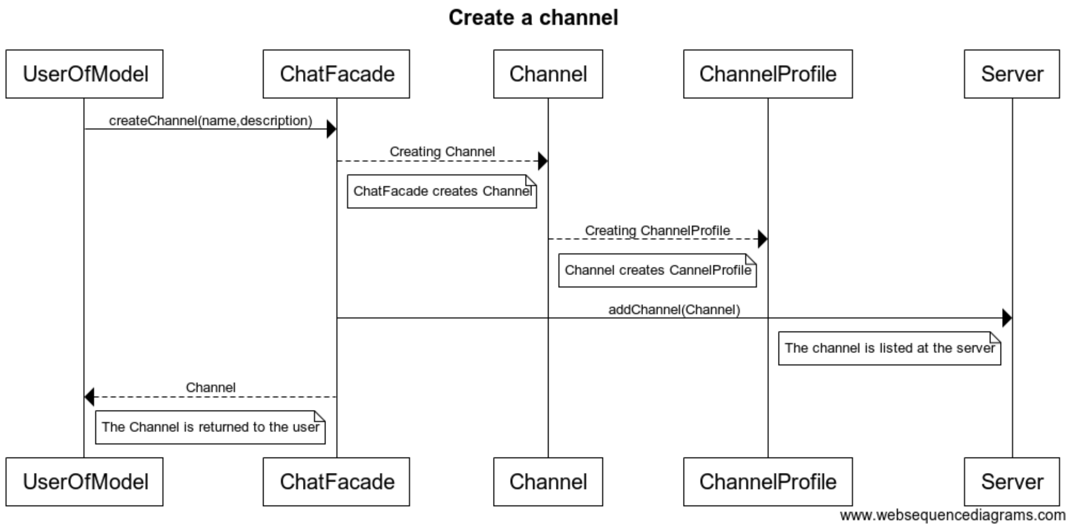
The message package is responsible for the message component. A message has a type, a sender, a timestamp and a content. The message package provides an factory for creating messages so other packages don't need to have dependencies on several different content classes.

The user package contains all data about the user. This package is dependent on the client package because it needs to be able to tell clients when a channel has been updated. Hence, when the user receives an update it will forward these to all it's clients. A user can have multiply clients, to support the ability to log in on the same user on different devices(or in our case, different windows).

The server package is responsible for keeping track of all channels and users. Every time a channel or user is created it is added to the servers lists. The server is also responsible for saving the data when the program closes. To do this, it uses some given data handler that implements IDataHandler. It gives this datahandler certain data objects that stores all the channel or user data as strings. When initializing the server, the given datahandler is used to get these data objects back and then the server turns them into real channel and user objects.

By breaking out the Identifiable and Recognizable interfaces into a separate package called identifiers, the amount of dependencies has been reduced. Furthermore, there were some circular dependencies that have been removed completely due to the package decomposition

Here is an example of how the model will create a channel.


Create a channel
UserOfModel
ChatFacade
Channel
ChannelProfile
Server
createChannel(name,description)
Creating Channel
ChatFacade creates Channel
Creating ChannelProfile
Channel creates CannelProfile
addChannel(Channel)
The channel is listed at the server
Channel
The Channel is returned to the user
www.websequencediagrams.com

As seen in the figure, the ChatFacade will create the channel, add it to the Server, and then return it to be used.

2.3 Controllers

This component takes care of all the communication between the view and the model. It handles all input the fxml-files receive and communicates with the model if needed. This package also listens to the model using the observer pattern and when the model tells this package that a new message has been sent, it will ask the model for the new message, and update the view according to the change. The controllers are structured in the way that the login view has its own separate controller and the actual chat window has a main controller, that uses several other smaller controllers which each is responsible for a special part of the window.

The controllers are dependent on the model's ChatFacade and the following interfaces:

- IChannel

- IUser
- IClient
- IClientListener
- IIdentifiable
- IRecognizable

Additionally, as the controller package is only dependent on the facade and some interfaces within the model, it tells us that it's quite loosely coupled and doesn't have to depend on concrete classes. And as said before, the controllers are not tightly coupled with the view either. The views are forwarding their input to some unknown object that implements the correct methods. The controllers don't know who they are receiving input from and consequently it's easy to change the associated fxml-file.

2.4 Services

The services package consists of two subpackages, password and datahandler.

The datahandler package is responsible for saving the programs data. To do this, it provides a concrete data handler class which implements an interface in the model. The controller package can then tell the model that this is the data handler to use. The model will then use this data handler, hidden behind the interface, to save data to a file and then later restore the data when its needed. When saving, the data handler is given certain data objects, that contains all the data of the components that are to be saved, as strings. When restoring, the model will ask the data handler to represent the data as these data objects and return them so that the model is able to recreate these data objects.

The password package...

3 Persistent Data Management

The application stores data by using JSON. When closing the application the application flushes all the data to a JSON-file. This makes it so that when we start the application again we can loop through the JSON-file and recreate how the data looked when we closed the application.

The data that we store are:

Channels, in the form of JSON-objects, that contains all the information that a channel contains, i.e. the users that are members of the channel and the messages of the channel.

Users, also in the form of JSON-objects, that contains the information of the user.

Images are stored locally and are accessed by using the path to the image.

4 Access Control and Security

In the application there is different kind of roles which changes some of the things a user can do in the application.

User: The most basic role anyone using the application can have. As soon as you login you become a user. The only thing a user can do is join channels and see other channels.

Channel member: As soon as you join a channel you become a channel member. As a channel member you can write and read messages that are written inside the channel. You also can add other members to the channel.

Channel administrator: If you create a channel you automatically become a channel administrator. A channel administrator can change the channel image, kick members from the channel and also all the functionality a ordinary channel member.

5 References